Teaching Test-Writing as a Variably-Scaffolded Programming Pattern

Nelson Lojo
Armando Fox
nelson.lojo@berkeley.edu
fox@berkeley.edu
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
Berkeley, CA, USA

ABSTRACT

Testing and test-writing are key skills for software engineers. Yet most CS curricula spend insufficient time on testing [16], and some studies have even found that graduating CS students enter developer jobs without these skills [5]. We argue that test writing is a *programming pattern* that, at least when going beyond simple input/output test cases, is new and unfamiliar to most students, so that teaching test-writing requires not only teaching the strategic concepts of thinking about testing, but also how to instantiate the "arrange–act–assert" pattern of which all tests consist. Teaching how to recognize and instantiate this pattern is complicated by having to learn how to use testing frameworks and libraries—essentially a syntactic obstacle. Faded Parsons Problems [25] (FPPs) have been shown to be a novel and effective type of exercise for exposing students to programming patterns by varying the amount of scaffolding provided. FPPs give similar learning gains to code-writing exercises but are preferred by students, and can be designed to "scaffold away" some syntactic obstacles that can impede students' ability to become fluent in test-writing. To our knowledge, neither FPPs nor original Parsons Problems [20] have previously been proposed to teach advanced programming patterns to advanced students. We present our design of a system for creating variably-scaffolded test-writing exercises in the form of FPPs, including autograding tests using existing autograding solutions augmented with techniques from mutation testing.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; • **Applied computing → Computer-assisted instruction**; **Learning management systems**;

KEYWORDS

Educational Tools, software testing, mutation testing, Parsons problems, PrairieLearn

ACM Reference Format:
Nelson Lojo and Armando Fox. 2022. Teaching Test-Writing as a Variably-Scaffolded Programming Pattern. In *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol. 1 (ITiCSE 2022), July 8–13, 2022, Dublin, Ireland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3502718.3524789

ITiCSE 2022, July 8–13, 2022, Dublin, Ireland.

ACM ISBN 978-1-4503-9201-3/22/07.
https://doi.org/10.1145/3502718.3524789

1 OVERVIEW AND CONTRIBUTIONS

Software testing typically takes 50% or more of the resources devoted to a software development project [15]. Yet teaching testing is under-emphasized in North American computer science classrooms, and when we do teach it, we are prone to using toy examples rather than realistic ones, simplified testing tools rather than standard ones, or both [16]. As a result, even some graduating seniors from CS lack basic testing proficiency [5], and it remains one of the most commonly cited technical skill gaps reported by employers [14, 22].

We argue that test-writing is a *programming pattern* and should be taught as such. Programming patterns (sometimes called plans, schemata, templates) have several subtly different definitions in the literature [3, 7, 17, 18, 24, 29, 31], but in general are higher-level, partial implementations of reusable higher-level concepts that achieve some goal. For example, an introductory-level programming pattern is "premature return from a loop": searching a collection until some element satisfies a predicate, then breaking out of the loop to immediately return that element, with a catch-all return after the loop in case no match is found. According to cognitive theory, when people view example problems with identifiable similarities, they eventually construct, store, and are able to recall and reuse a complex pattern when solving a problem that fits the pattern [6, 17], rather than constructing the problem's solution from scratch. Learning to recognize and apply patterns is therefore critical to becoming a proficient software engineer.

If test-writing can be cast as a programming pattern, then it should be possible to teach it using pedagogical tools designed to scaffold the learning of patterns. Specifically, we can scaffold away some of the syntactic and mechanical obstacles that are known to arise when trying to master a new pattern generally [31] and testing-related skills specifically [21], allowing the student to first master the conceptual aspects of the pattern, and then gradually fade the scaffolding and move the student towards writing tests from scratch with the correct syntax.

The contributions of this paper are therefore as follows:

- We propose that test-writing can be formulated as an example of a *programming pattern* that is unfamiliar to early-career students.
- We identify both conceptual and syntactic obstacles associated with learning to instantiate this pattern when the testing framework and/or the system under test are nontrivially complex, as is the case with real-world applications.

- We propose how these obstacles can be addressed by Faded Parsons Problems (FPPs) [26], a new type of machine-gradable exercise that helps teach patterns by having students reconstruct expert solutions with varying amounts of scaffolding.
- We demonstrate a methodology and an instructor-facing tool for authoring and delivering FPP-based test-writing exercises. Our approach enables automated grading of student work using a technique based on mutation testing.

Parsons Problems have traditionally been used primarily to teach introductory programming patterns to novice students; to our knowledge, this is the first attempt to use them to teach a syntactically complex programming pattern to advanced students.

We also note two non-contributions. First, we are just now in the process of deploying these tools in our own courses. Evaluating the efficacy of our approach and tools on student learning will be the subject of a future paper. Second, good testing discipline encompasses many topics besides test-writing, such as defining and capturing test coverage metrics, identifying critical values for test case parameters, and teaching the process of test-driven development (TDD). While we are supportive of these goals, they are orthogonal to (and so outside of) the work described in this paper, though we plan to use our approach as the vehicle for teaching those topics in our more advanced courses.

2 RELATED WORK

One way the behavior of expert programmers differs from that of novices is in experts' ability to leverage patterns in understanding and writing code [23, 29]. Attempts to incorporate the teaching of patterns/templates into computer science classes [18, 31] led to novices improving their problem decomposition and solution construction skills (i.e. coding ability). Hence our motivation for attempting to frame test-writing as a programming pattern.

Benefits and challenges of teaching testing. Edwards and others [4, 10] have extensively studied and confirmed the benefits of teaching software testing, including breaking student dependence on debugging using only instructor-provided tests. But Proulx confirms [21] that syntactic and mechanical obstacles do indeed arise when using industrial-grade testing tools in the classroom, and offers a simplified testing tool and a library that uses Java reflection to automate some of the work required to create fully fleshed out test cases. In contrast, we use a real (not simplified) testing tool in our exercises, relying instead on the scaffolding provided by Faded Parsons Problems to temporarily "get the syntax obstacle out of the way" by focusing on reconstructing an expert's solution.

Parsons and Faded Parsons Problems for teaching patterns. Regular Parsons Problems [20] ask students to rearrange scrambled lines of code to reconstruct a solution; in this sense, they focus more on code reading than code writing. Faded Parsons Problems [25] allow introducing blanks into some or all of the given lines of code, gradually shifting the focus towards code writing. By fading the scaffolding over time, we aim to provide students a continuous "on-ramp" to learning to write complex tests from scratch. While regular Parsons Problems provide similar learning gains to code-writing exercises while taking less time [11, 12], those learning gains do not necessarily transfer to code-writing tasks [25]. In contrast, Faded Parsons Problems were specifically found to improve code-writing skills as well as or better than code-writing exercises [26]. We therefore use Faded Parsons Problems (hereafter FPPs) for our approach.

Mutation testing to automate grading of student work. Software testing is a rich field with no shortage of techniques for creating and analyzing test suites [2]. Our approach to grading student tests is based on Ammann and Offutt's *mutation testing*, in which bugs are deliberately introduced into code to verify that some test fails as a result. Others have used mutation testing to help teach test-writing; Clegg et al. [8] "gamify" software testing by having an attacker create mutants and a defender write test cases that detect them, but in general the tests they illustrate are limited to pure functions with no side effects or collaborators. We are interested in scaffolding the writing of more advanced and realistic tests that rely on nontrivial testing libraries to support the activities of arranging, acting, and asserting.

Wrenn and Krishnamurthi [30] observe that when the same student writes the code and the tests, the same misunderstandings may permeate both. They use mutation testing to provide students with instant feedback on their test cases, independent of their code implementation progress. We go further and propose that as an "on-ramp" to test-writing, students first reconstruct variably-scaffolded expert-curated solutions (test cases) prior to constructing them from scratch. Variable scaffolding also ensures that students cannot "game" the process of reconstructing the solution based on cues such as data dependencies [9]. The same authors propose scaffolding the task of test-writing based on the six-step "design recipe" in *How To Design Programs* [13], which specifies a series of steps with concrete deliverables rangingfrom requirements-understanding to implementation of each test case. We instead scaffold the *syntactic and mechanical* aspects of test-writing by having students reconstruct expert code in which these mechanical details have been accounted for.

3 WHAT MAKES LEARNING TEST-WRITING HARD?

Software testing is a critically important field with a rich literature [2]. Far from being of only academic importance, good testing is so critical to complex commercial software that two of the world's largest and most successful software companies have published books focusing exclusively on their testing methodologies, tools, and approaches [19, 28]. Yet in our own curriculum, testing is covered only briefly in our introductory CS series. Students do not encounter advanced testing techniques (test doubles, method stubs and spies, mock objects, and so on) until the advanced courses. We next describe the difficulties students have when they first encounter these advanced techniques, and how framing test-writing as a pattern can help overcome them.

3.1 Test Writing as a Pattern

Thinking about test-writing differs in important ways from thinking about code, because it often involves finding ways to circumscribe and control specific behaviors of the system under test (SUT). For example, when testing a method M1 that calls another method M2, it is desirable to control the behavior of M2, both to test M1's

```
1  specify 'centenary years are not leap years' do
2    expect(leap_year? 1900).to be_false
3  end
4  specify 'but quadricentennials are leap years' do
5    expect(leap_year? 2000).to be_true
6  end
```

```
1  specify 'search user by email succeeds if user exists' do
2    user = create('user', email: 'macallan@fakemail.com')
3    search_result = User.search('macallan@fakemail.com')
4    expect(result).to eq(user)
5  end
```

Figure 1: (a) Top: two one-line test cases with no Arrange step, and with the Assert and Act steps combined in a single line of code. (b) A test requiring an explicit Arrange step. The syntax is RSpec (rspec.info), a widely-used Ruby test framework, augmented by the FactoryBot library, which provides an object factory for test suites.

behavior in different cases (what if M2 returns an error or raises an exception?) and to isolate the testing of M1 from the testing of M2 (so that actual bugs in M2 do not also spuriously cause test failures on M1). Learning to think in this way is the *conceptual* hurdle of advanced test-writing.

Real-world testing frameworks provide quite a bit of machinery to handle testing scenarios such as the above (in this example, creating a test double or stub method for M2), and this machinery has mechanics and syntax of its own that students must master, of which we give examples in the next section. This is a *syntactic* hurdle.

These two hurdles—conceptual and syntactic—are exactly the ones that have been effectively addressed by teaching programming patterns. The idea is to scaffold away the syntactic hurdles initially, so that the student can concentrate on learning the conceptual ones. For example, blocks-based programming languages like Scratch and Blockly make most syntax errors impossible. Once the student becomes comfortable with the concept, the scaffolding can gradually be faded so that the student can begin mastering the syntax necessary to instantiate the pattern.

We next describe the pattern that underlies substantially all test-writing, give specific examples of these conceptual and technical hurdles, and describe our proposal to use Faded Parsons Problems to support the teaching of that pattern.

3.2 The Arrange-Act-Assert Pattern and Its Conceptual and Syntactic Hurdles

The basic pattern followed by all automated software tests is Arrange, Act, Assert [15]. The Arrange step sets up any necessary preconditions, such as state externally visible to the system under test (SUT), use of test doubles or spies, and so on. The Act step stimulates the system under test, which may be a leaf function, a function that calls other functions, and so on. The Assert step checks that the certain postconditions are met as a result—for example, a certain value being returned, or certain events occurring or not occurring as a result of exercising the SUT. The complexity of each of the three steps—Arrange, Act, Assert—varies depending on the test.

```
1  class UsersController < ApplicationController
2    def search_for_user
3      search_string = params['string']
4      @user = User.search(search_string.downcase)
5      render(template: 'not_found') and return if @user.nil?
6    end
7  end
```

```
1  specify 'search receives correct params and sets result' do
2    fake_user = double('User')  # arrange
3    allow(User).to receive('search').
4                    with('macallan@fakemail.com').
5                    and_return(fake_user)   # arrange
6    get('search_for_user', :string => 'MacAllan@fakemail.com')
7    expect(assigns('user')).to eq(fake_user)
8  end
```

Figure 2: (a) A snippet of code from a Web app, (b) a test for that code that uses a method stub and mock object (lines 2–5) to isolate the test from the dependency on `User.search`, and uses RSpec library functions `get` (Act step) and `assigns` (Assert step).

In the simple unit tests most students encounter in introductory CS courses, the Arrange step does nothing, the Act step calls a function with some specific argument values, and the Assert step checks the return value. Figure 1(a) shows an example. But consider an only slightly more complex test case that might appear in the test suite for a Web application using Ruby on Rails [1]. The test in Figure 1(b) checks that the static method `search` correctly finds and returns a user with a particular email if that user exists. The *conceptual* hurdle of formulating this test case is realizing that before we stimulate the SUT (Act step, line 3), we must first create a fictitious user for it to find (Arrange step, line 2), and then verify the created user was found (Assert step, line 4). The *syntactic* hurdle is that the student must know that the FactoryBot testing library provides the library call `create`, which invokes an object factory to create valid instances on demand of a desired class for use by test cases.[2]

The above syntactic hurdle may seem modest, but now consider a slightly more complex example for the same fictitious app. Figure 2(a) shows a simple piece of Rails code that would be invoked when someone submits a Web form to search for a user by their email address. The corresponding test in Figure 2(b) checks that when the user types a string into a Web form with the goal of searching for a user name matching that string, the result of the search is correctly made available to be displayed as part of an HTML response page.

This test introduces a new *conceptual* hurdle. Since the behavior of `User.search` is covered by tests such as that in Figure 1(b), the goal of the test in Figure 2(b) is to verify that all the "glue" around the call to `User.search` works correctly. In this case, good testing practice calls for *insulating* the test from the implementation of `User.search` by using a method stub. The corresponding *syntactic* hurdle is in knowing how to set up the stub (lines 3–5) and create a mock object for the stub to return (line 2).

[1] https://rubyonrails.org

[2] Not to be confused with a factory object or with the Abstract Factory software pattern, an *object factory* in object-oriented programming is an object for creating other object instances.

An additional *syntactic* hurdle results from the test exercising a full-stack Web application, rather than simply calling a method. In line 6, the call to `get()` is an RSpec extension that handles the non-trivial work of simulating how a well-formed HTTP request resulting from the form submission would be parsed, including extracting the "user-entered" parameter value `string=macallan@gmail.com` from the URI, and creating the data structure that would normally be passed to the Rails code of Figure 2(a). In line 7, the call to `assigns()` inspects the list of variables that are about to be made available to the view-template renderer. Neither of these functions is *conceptually* central to the test itself, but it is necessary to get their syntax correct in order for the test to run.

In summary, although both examples consist of short tests that clearly follow the arrange–act–assert pattern, *understanding how to set up preconditions* creates new conceptual hurdles for the student, and *knowing how to formulate the corresponding library calls* presents a syntactic hurdle. Note that even if the student knows the programming *language's* syntax, there are plenty of legal-but-incorrect ways to call these specific "testing library" methods.

We next describe our tool that uses Faded Parsons Problems to scaffold the learning of these conceptual and syntactic hurdles.

4 TEST WRITING AS A FADED PARSONS PROBLEM

As an alternative to writing code from scratch, Faded Parsons Problems [25], recently introduced by Weinman et al., provide a set of scrambled lines of code, some of which may include blanks. Students fill in the blanks and reorder the lines to reconstruct an intentionally-designed solution to a programming task.

Our goal is to enable instructors to create autograded test-writing exercises as FPPs. Our approach preserves the FPP benefit of introducing students gradually to a new programming pattern by removing some cognitive load associated with syntax and mechanics. Furthermore, using FPPs prevents "gaming" test-writing by creating a test case that trivially passes (or fails) without exercising the system-under-test. With PPs and FPPs, the student is restricted to using the lines of code they are given.

4.1 Autograding Approach

Before describing the instructor and student view of using our tool, we describe our approach to autograding, which determines what materials the instructor must author. Code autograders for student programs typically work by running instructor-provided test cases on student-submitted code. But when the student code itself consists of test cases, how should the autograder work?[3]

Mutation testing [2] introduces deliberate, syntactically-legal defects in the code being tested; if no test case fails as a result, a gap in test coverage is indicated. We use a variant of mutation testing similar to that used in [30], some of whose terminology we follow here. A suite of one or more tests (whether student-authored or instructor-supplied) is *valid* if its assertions pass when run against a correct implementation of the code. It is *thorough* if it detects all defects in a buggy implementation. In our approach, the instructor provides a correct implementation of the system under test (SUT), a valid and thorough *reference suite* of tests for the SUT, and a set

[3]To abuse a cliché, *quis probet ipsos probationes?*

```
1  def leap_year(year)
2    if (year % 4 != 0)       # mutant 1: year % 4 == 0
3      false
4    elsif (year % 100 == 0)  # mutant 2: year % 100 != 0
5      (year % 400 == 0)      # mutant 3: year % 400 != 0
6    else
7      true
8    end
9  end
```

```
1  solution: |
2    describe 'leap year' do
3      specify 'occurs every 4 years' do
4        ?expect?(leap_year(2004)).to be_?truthy?
5      end
6      specify 'occurs on quadricentennial years' do
7        expect(leap_year(?2000?)).to be_?truthy?
8      end
9      specify 'does not occur on other centennials' do
10       expect(leap_year(?1900?)).to be_?truthy?
11     end
12   end
13 mutations:
14   var_logic_short_circuit:  # variant 1
15     leap_year.rb:
16       2-3: |
17             if (year % 4 == 0)
18   var_centennials:          # variant 2
19     leapYear.rb:
20       4-5: |
21             if (year % 100 != 0)
22   var_400_year_check:     # variant 3
23     leapYear.rb:
24       5-6: |
25             (year % 400 != 0)
```

Figure 3: Instructor provided assets
Top (a): a simple Ruby function and possible mutations. Bottom (b): The instructor provides a question prompt, a reference suite, and a set of suggested mutations. Question marks in the reference suite indicate where blanks would appear in the FPP.

of manually-constructed mutants, in each of which some deliberate bug has been introduced that would cause one or more tests in the reference suite to fail.

The student is shown the SUT and a set of scrambled code lines (some possibly containing blanks) that they must reconstruct into a test suite. We run the student's suite and reference suite side-by-side for the SUT and for each mutant. We say that a student-suite test and the corresponding reference-suite test *match* with respect to a mutant if they exhibit the same behavior (passing or failing) when run against that mutant. The student's score is computed as a suitable aggregate across all mutants and all tests; the appropriate rubric would depend on how the exercise is being used (summative vs. formative assessment, for example), a discussion of which is beyond the scope of this paper.

Intuitively, if some mutation is introduced specifically to expose a particular test condition, then there should be a test in the reference suite that checks that condition and fails when run against the mutant. Therefore, a correct student test suite should include a test that behaves in the same way.

4.2 Instructor's View

Consider the simple instructor-authored SUT in Figure 3(a), a function that tests whether a particular year in the Gregorian calendar

is a leap year.[4] Lines 2, 4, and 5 are annotated to show how to create mutants that would detect a non-thorough student test suite. For example, changing `==` to `!=` in line 5 (reversing the sense of the test) would create a mutant that fails when specifically checking a quadricentennial year. A thorough student test suite would therefore need to include at least one test case that checks this behavior. Note that other mutants are also possible, such as changing line 3 from `false` to `true`. In Section 6 we describe the possibility of automatically generating mutants using existing technology.

Figure 3(b) shows additional instructor-provided materials including a solution (reference suite) and mutations. The solution may, but does not need to, selectively place question marks (?) around certain tokens; if so, those tokens will appear as blanks when the student is shown the scrambled code lines.

The description of mutations[5] is used to create separate mutants of the SUT, one per mutation. The student's submission and reference suite will each be run against the SUT and against each mutant; 100% correct means that the student and reference suites behave identically on the SUT and on all mutants.

4.3 Student's View

For those unfamiliar with Faded Parsons Problems, a short demo video by its inventors, published alongside the original paper [26], can be seen at youtu.be/dmcCxy34NE8 and shows the general student experience of working on this problem type, using a standalone system built to deliver FPPs.

We build our implementation using PrairieLearn [27], an open-source and extensible assessment authoring system developed primarily at the University of Illinois at Urbana-Champaign, recent extensions to which allow constructing and autograding Faded Parsons Problems [1]. While our methodology is language-independent, and adding new languages is straightforward (PrairieLearn has excellent support for this by using Docker containers), our initial implementation uses Ruby and the RSpec testing framework due to curricular demands of specific courses.

The screenshot in Figure 4 shows the student's view immediately after submitting a candidate solution to a subset of the above example problem. The feedback shows that the student's test case passes against the SUT (as expected) and fails against a particular mutant that inverts the sense of the test in line 2 of Figure 3 (also as expected).

Creating a complete autogradable FPP in PrairieLearn requires a set of about a dozen files arranged and formatted in a specific way; our tooling ingests the instructor-authored SUT and the YAML file indicating the question prompt, mutants, and other options, and creates the necessary file structure for PrairieLearn. For example, our tool creates each mutant by applying the specified mutations to a copy of the SUT one at a time, labels each mutant uniquely so that its output in the test results can be distinguished from that of other mutants, arranges to concatenate the submitted student tests and/or instructor tests with the SUT and mutants, and parses the output of the code autograder to score the matching behavior between the student's and the instructor's suite.

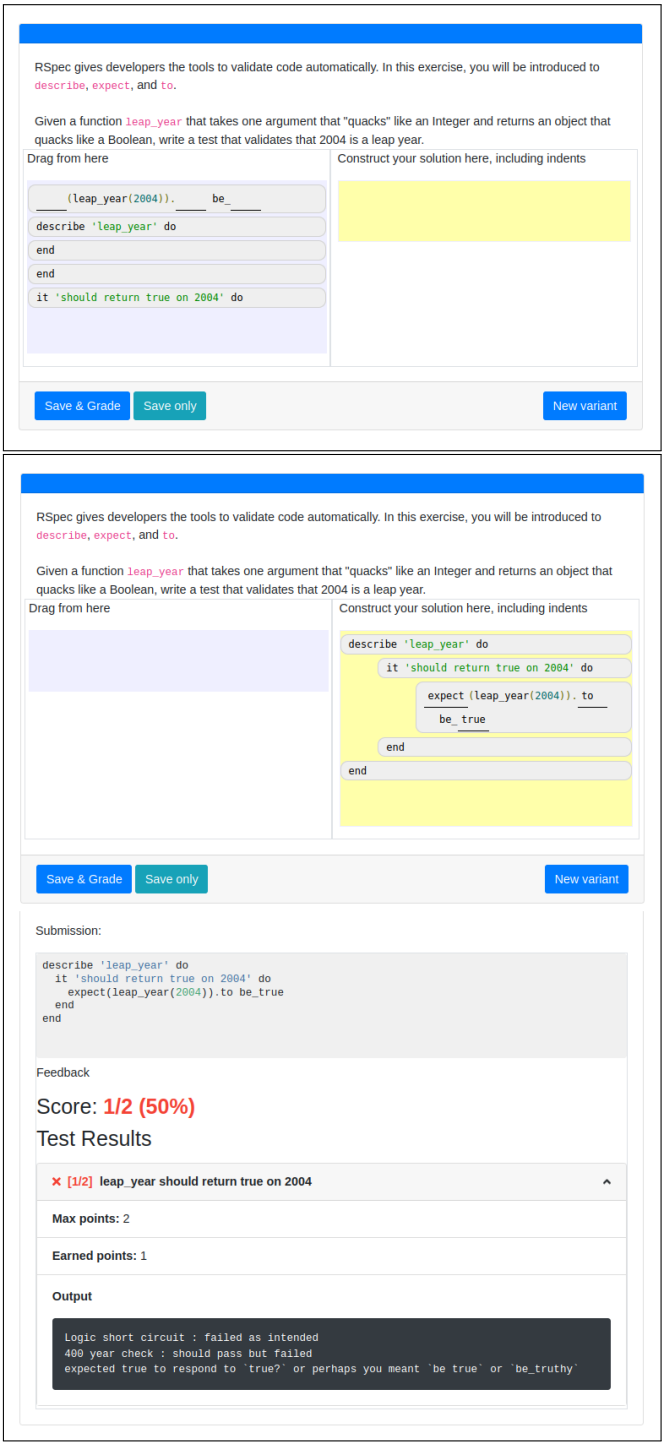

Figure 4: Student's view of solving an FPP
Top: initial screen includes a question prompt, a tray of scrambled code lines with some containing blanks, and an empty tray into which the student drags lines in the correct order and fills in the blanks. Bottom: A submission attempt in which the student made a syntax error filling in the blank. The student was shown the full SUT code on a previous screen.

[4] A leap year occurs every four years, but does *not* occur in centennial years such as 1900, *unless* it is also a quadricentennial (every 400 years) such as 1600 or 2000.
[5] We are modifying this notation to use the `patch(1)` format.

We are making our PrairieLearn element code and related tooling available under an open source license at github.com/ace-lab/pl-ucb-faded-parsons.

5 DISCUSSION

A few observations based on the example are worth pointing out. First, if the student were writing test code from scratch, they could "game" the scoring of test cases by providing cases that make trivial assertions such as `expect(1).to eq(0)`. In our methodology, the provided lines of code may contain blanks that can be filled in, but the lines cannot be replaced. That is, since students are *reconstructing* an expert's solution (the reference suite), they are constrained to using the lines of code provided to them.

Similarly, if the expected behavior is for a test to fail, the student only gets credit if the test fails because its assertion(s) failed, not because of (e.g.) syntax or other runtime error in the test, such as might be caused if the student filled in a Faded Parsons blank with a value that would cause a runtime type error.

A valid and thorough student suite might not be identical to the instructor's reference suite. For example, `expect(val).to be_truthy` is an alternative (and more idiomatic) way to assert "truthiness" in Ruby. If the FPP were presented such that the tokens `eq(true)` (in lines 11, 14, or 17 of Figure 3b) were surrounded by question marks, these would appear as blanks that the student has to fill in. It would be syntactically legal for the student to fill them in with `be_truthy`, and would result in equally valid and thorough tests.

Finally, although this example is a simple pure-function case in which a unit test simply checks whether a set of known inputs produces given outputs, tests may also require using test doubles (mocks and stubs) to isolate the unit test's behavior from collaborator classes or methods, asserting the presence of absence of exceptions, test spies to check whether a given external method was called, and so on.

6 LIMITATIONS AND FUTURE WORK

In our initial version, tests cannot be weighted differently (so that passing some tests is worth more points than passing others). As we integrate this tool into our pedagogy, we will use the needs of grading rubrics as a guide to providing more flexible scoring options.

Our initial version allows specifing n mutations and produces n mutants from the SUT, each containing a single mutation. That is, it is currently not possible to combine multiple mutants into a single test run. We view this as an acceptable limitation since our goal is *not* to use this methodology to raise students' awareness of the power of mutation testing, but simply to allow enough autograding for student work products that are themselves test cases.

Perhaps the biggest current limitation is that the entire "suite" to be provided by the student must take the form of a single FPP. For example, the reference solution in Figure 3b has 11 lines of test code, all of which would be present (but scrambled) in the Faded Parsons problem for the student. Pedagogically, it might be preferable (for example) to first have the student author a single test case, then continue on to author the other two. Students may submit a suite with fewer test cases than the reference suite by not dragging all the code lines from the code line tray into the solution tray; in this case, "missing" tests are always counted as not matching the corresponding reference-suite test, but ideally, the student would first be scaffolded through writing correct and complete test cases for a subset of conditions, and then cumulatively add to that suite through subsequent exercises. We are exploring how to achieve this given some of the limitations of the PrairieLearn framework on which our tool is based.

Finally, other researchers [8] have used automatic static-analysis and test-generation tools to help automatically generate mutants for use in teaching test-writing. We plan to apply this idea in our approach, to minimize the amount of manual work instructors must do at exercise-design time.

7 CONCLUSION

We have presented the challenge of overcoming syntactic and mechanical obstacles to test-writing as a problem of becoming comfortable with a new programming pattern, and therefore proposed the use of a relatively new exercise type, Faded Parsons Problems, which have been shown to be effective in helping students learn to both recognize and use new programming patterns and in providing a smooth on-ramp to writing code that uses those new patterns from scratch. We believe this is the first such approach to teaching test-writing in this scaffolded manner, and the first application of (a variant of) Parsons Problems to teaching advanced post-CS1 concepts. Our tooling uses mutation testing as the basis of autograding the student's test cases, and is available as an extension to the PrairieLearn open-source assessment framework. We look forward to using it to help drill more advanced test-writing skills in our upper division programming courses.

ACKNOWLEDGMENTS

Ajay Vellayapan also contributed to the design and implementation of the RSpec/Faded Parsons extensions to PrairieLearn. This work was supported by grant #OPR19186 from the California Education Learning Lab (calearninglab.org), an initiative of the California Governor's Office of Planning and Research.

REFERENCES

[1] [n. d.]. (In submission, blinded for review).
[2] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press. http://www.amazon.com/Introduction-Software-Testing-Paul-Ammann/dp/0521880386
[3] Owen Astrachan, Garrett Mitchener, Geoffrey Berry, and Landon Cox. 1998. Design patterns: an essential component of CS curricula. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education - SIGCSE '98*. ACM Press, New York, NY, USA, 153–160. https://doi.org/10.1145/273133.273182
[4] Kevin Buffardi and Stephen H. Edwards. 2015. Reconsidering Automated Feedback: A Test-Driven Approach. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 416–420. https://doi.org/10.1145/2676723.2677313
[5] Jeffrey C. Carver and Nicholas A. Kraft. 2011. Evaluating the testing ability of senior-level computer science students. In *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, Honolulu, HI, USA, 169–178. https://doi.org/10.1109/CSEET.2011.5876084
[6] William G. Chase and Herbert A. Simon. 1973. Perception in chess. *Cognitive Psychology* 4, 1 (Jan. 1973), 55–81. https://doi.org/10.1016/0010-0285(73)90004-2
[7] Michael J. Clancy and Marcia C. Linn. 1999. Patterns and pedagogy. *ACM SIGCSE Bulletin* 31, 1 (March 1999), 37–42. https://doi.org/10.1145/384266.299673
[8] Benjamin S. Clegg, Jose Miguel Rojas, and Gordon Fraser. 2017. Teaching Software Testing Concepts Using a Mutation Testing Game. In *2017 IEEE/ACM*

39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET). IEEE, Buenos Aires, 33–36. https://doi.org/10.1109/ICSE-SEET.2017.1

[9] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a New Exam Question: Parsons Problems. In *Proceeding of the fourth international workshop on Computing education research - ICER '08*. ACM Press, New York, NY, USA, 113–124. https://doi.org/10.1145/1404520.1404532

[10] Stephen H. Edwards. 2004. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. *SIGCSE Bull.* 36, 1 (mar 2004), 26–30. https://doi.org/10.1145/1028174.971312

[11] Barbara J. Ericson, James D. Foley, and Jochen Rick. 2018. Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, New York, NY, USA, 60–68. https://doi.org/10.1145/3230977.3231000

[12] Barbara J. Ericson, Lauren E. Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research - Koli Calling '17*. ACM Press, New York, NY, USA, 20–29. https://doi.org/10.1145/3141880.3141895

[13] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs, second edition: An Introduction to Programming and Computing (The MIT Press)* (paperback ed.). The MIT Press. 792 pages. https://htdp.org

[14] Armando Fox and David Patterson. 2012. Viewpoint: Crossing the Software Education Chasm. *Commun. ACM* 55, 5 (5 2012), 25–30.

[15] Armando Fox and David Patterson. 2020. *Engineering Software as a Service: An Agile Approach Using Cloud Computing, Second Edition*. Pogo Press.

[16] Vahid Garousi and Aditya Mathur. 2010. Current State of the Software Testing Education in North American Academia and Some Recommendations for the New Educators. In *2010 23rd IEEE Conference on Software Engineering Education and Training*. IEEE, Pittsburgh, PA, USA, 89–96. https://doi.org/10.1109/CSEET.2010.29

[17] Sandra P. Marshall. 1995. *Schemas in Problem Solving*. Cambridge University Press, New York, NY, USA. https://doi.org/10.1017/cbo9780511527890

[18] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '07*. ACM Press, New York, NY, USA, 151–155. https://doi.org/10.1145/1268784.1268830

[19] Alan Page, Ken Johnston, and BJ Rollison. 2008. *How We Test Software at Microsoft* (1st edition ed.). Microsoft Press.

[20] Dale Parsons and Patricia Haden. 2006. Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) *(ACE '06)*. Australian Computer Society, Inc., AUS, 157–163.

[21] Viera K. Proulx. 2009. Test-driven design for introductory OO programming. In *Proceedings of the 40th ACM technical symposium on Computer science education - SIGCSE '09*. ACM Press, Chattanooga, TN, USA, 138. https://doi.org/10.1145/1508865.1508919

[22] Alex Radermacher and Gursimran Walia. 2013. Gaps between industry expectations and the abilities of graduates. In *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*. ACM Press, Denver, Colorado, USA, 525. https://doi.org/10.1145/2445196.2445351

[23] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education* 13, 2 (June 2003), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

[24] James C. Spohrer and Elliot Soloway. 1986. Novice mistakes: are the folk wisdoms correct? *Commun. ACM* 29, 7 (July 1986), 624–632. https://doi.org/10.1145/6138.6145

[25] Nathaniel Weinman, Armando Fox, and Marti Hearst. 2020. Exploring Challenging Variations of Parsons Problems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, USA, 1349. https://doi.org/10.1145/3328778.3372639

[26] Nathaniel Weinman, Armando Fox, and Marti A. Hearst. 2021. Improving Instruction of Programming Patterns With Faded Parsons Problems. In *2021 ACM CHI Virtual Conference on Human Factors in Computing Systems (CHI 2021)*. Yokohama, Japan (online virtual conference).

[27] Matthew West, Nathan Walters, Mariana Silva, Timothy Bretl, and Craig Zilles. 2021. Integrating Diverse Learning Tools using the PrairieLearn Platform. In *Seventh SPLICE Workshop at SIGCSE*.

[28] James Whittaker, Jason Arbon, and Jeff Carollo. 2012. *How Google Tests Software* (1st edition ed.). Addison-Wesley Professional.

[29] Susan Wiedenbeck, Vikki Fix, and Jean Scholtz. 1993. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies* 39, 5 (Nov. 1993), 793–812. https://doi.org/10.1006/imms.1993.1084

[30] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. ACM, Toronto ON Canada, 131–139. https://doi.org/10.1145/3291279.3339416

[31] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J. Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (Jan. 2019), 205–253. https://doi.org/10.1080/08993408.2019.1565235